Exception handling in Java 10

Usually we believe that a compiled program is error free and will always execute successfully, but in few cases the program can terminate while it is executing. For example, if we have written a program that connects to a particular website and download the web pages, under normal conditions the program will execute as expected but suppose if the program is executed in a computer where there is no internet connection then the program will produce unexpected output. There can be similar situations when we are dealing with files, for instance the program tries to modify a read-only file or it tries to open a file that does not exist in the system. Such cases are known as "exceptions" in Java. The exception results in an abnormal execution and it may lead to abnormal termination of the program.

An exception is an indication of a problem that occurs during a program's execution, it usually signals an error. Although exceptions occur infrequently, we must be careful to handle such cases while writing the code. Exception handling allows a program to continue executing as if no problem had been encountered or it may notify the user of the problem before terminating in an uncontrolled manner.

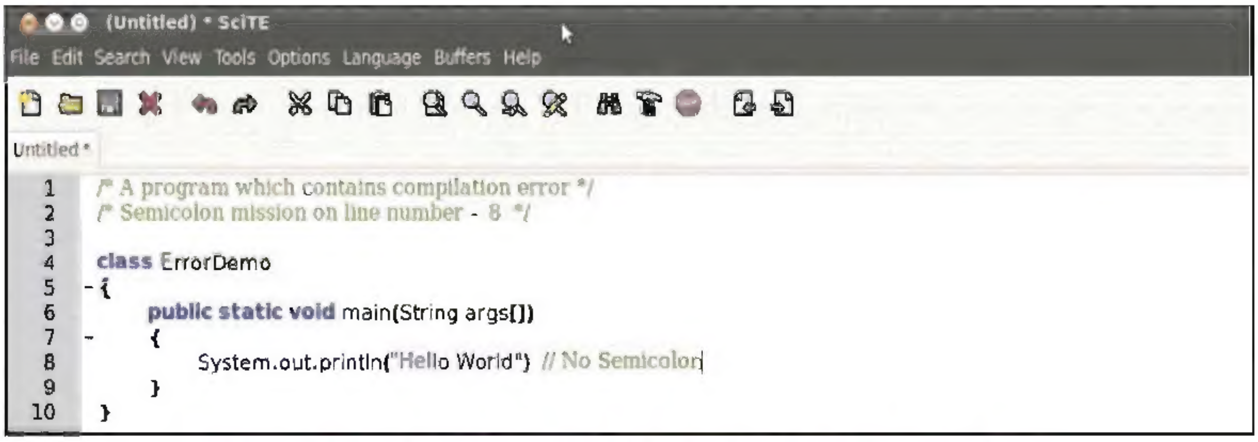
In this chapter we will learn techniques to handle exceptions in our programs, few standard exceptions available in Java, a technique to guarantee that a particular block of code will always be executed, even if exceptions are present in our program. Finally we will look at a technique to define and use our own type of exception

Types of Exceptions

In Java, all kinds of error conditions are called exceptions. Errors can be broadly classified into two categories namely Compile-time errors and Run-time errors.

Compile-time errors

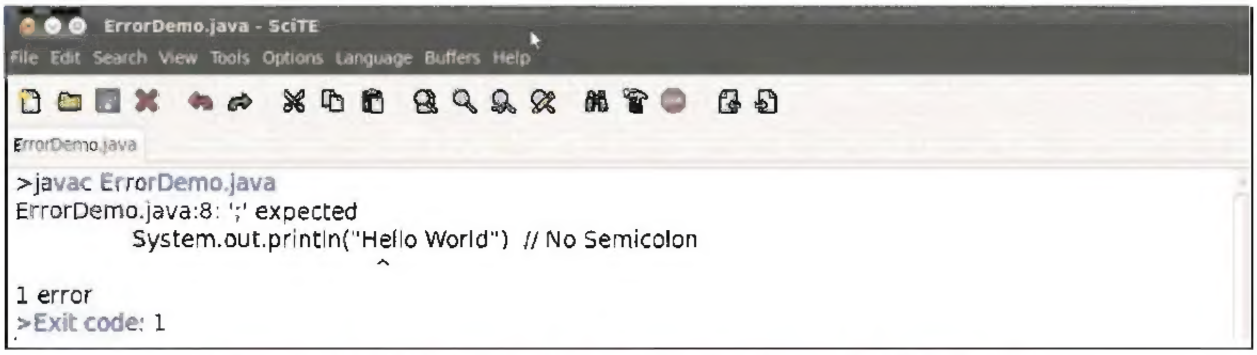
We have already learnt in previous chapters how to compile our java programs. A compiler is used to convert source code into object code. If there is a syntax error in the program we will get a compilation error and will not be able to create the ".class" file. Examples of some common syntax errors are missing semicolon, use of undeclared variable, wrong spellings of identifier or keyword and mismatch of bracket. The java program shown in figure 10.1 does not contain a semicolon.



```
/* A program which contains compilation error */
/* Semicolon mission on line number - 8 */

class ErrorDemo
{
    public static void main(String args[])
    {
        System.out.println("Hello World") // No Semicolon
    }
}
```

Figure 10.1 : A program that illustrates Compile-time Error

The above code when compiled will generate a compile-time error. This happens because we have missed writing the semicolon ';' in line number 8. The Java compiler when showing the output suggests the type of error, along with the line number where the error has occurred as shown in figure 10.2.



```
>javac ErrorDemo.java
ErrorDemo.java:8: ';' expected
        System.out.println("Hello World")  // No Semicolon
                                         ^
1 error
>Exit code: 1
```

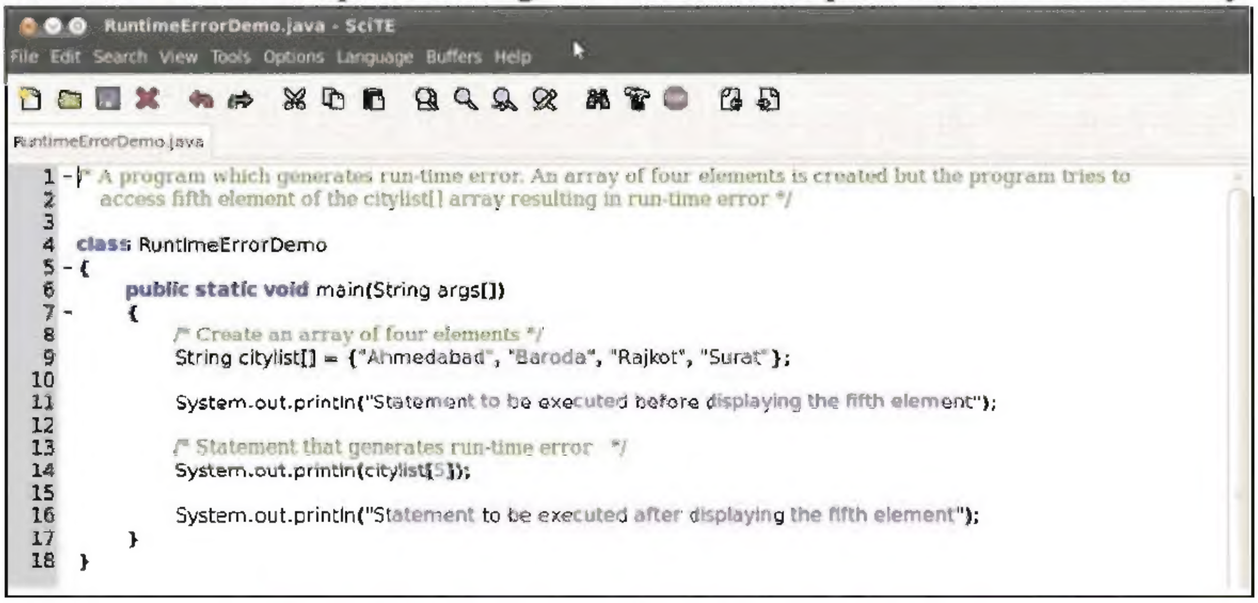
Figure 10.2 : Output of the program shown in Figure 10.1

Compile-time errors are usually the mistakes of a programmer and it won't allow the program to compile unless they are solved.

Note : In the field of Computer Science, "Exit code" or "Exit status" indicates whether the command or a program executed successfully or not. Code "0" indicates that the command executed successfully whereas code "1" indicates that some problem occurred while executing the command. In figure 10.2, the last line indicates "Exit Code: 1", it means that the compilation of program - ErrorDemo.java was not successful.

Run-time errors

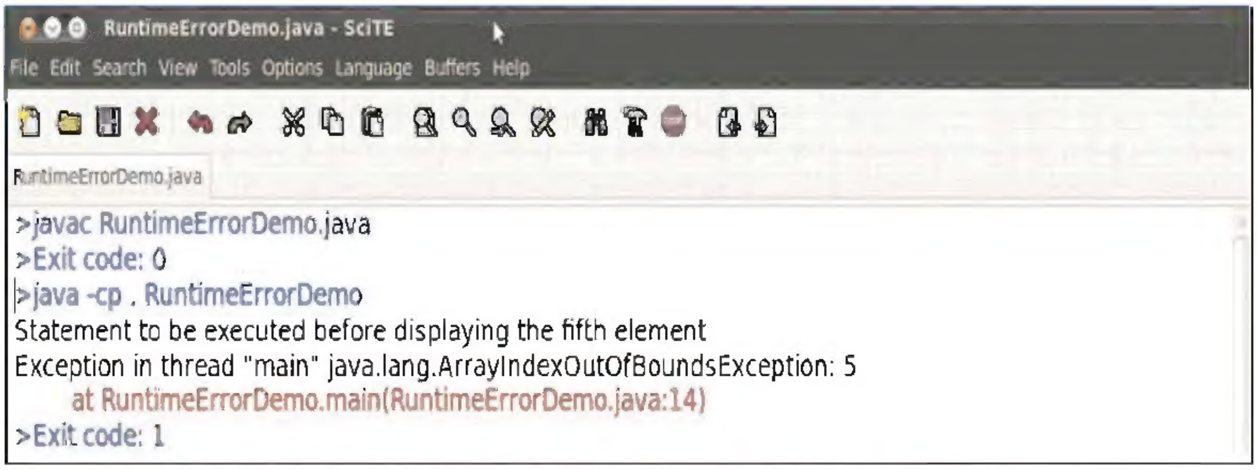
If there are no syntax errors in the source code then the program will compile successfully and we will get a ".class" file. However this does not guarantee that the program will execute as per our expectations. Let us look at another example shown in figure 10.3 which will compile but will terminate abnormally.


RuntimeErrorDemo.java - SciTE
File Edit Search View Tools Options Language Buffers Help
RuntimeErrorDemo.java

```
1  /* A program which generates run-time error. An array of four elements is created but the program tries to
2     access fifth element of the citylist[] array resulting in run-time error */
3
4  class RuntimeErrorDemo
5  {
6       public static void main(String args[])
7       {
8            /* Create an array of four elements */
9            String citylist[] = {"Ahmedabad", "Baroda", "Rajkot", "Surat"};
10
11           System.out.println("Statement to be executed before displaying the fifth element");
12
13           /* Statement that generates run-time error   */
14           System.out.println(citylist[5]);
15
16           System.out.println("Statement to be executed after displaying the fifth element");
17      }
18 }
```

Figure 10.3 : A program illustrating run-time error

The program shown in figure 10.3 will compile as there are no syntax errors it. In the program, we have created an array "citylist[]" that contains name of four different cities. In line 14 we are trying to display the contents of element of citylist array that does not exist. This case here is an exception condition. The exception will be generated during runtime; the output of the execution of the program is shown in figure 10.4.


RuntimeErrorDemo.java - SciTE
File Edit Search View Tools Options Language Buffers Help
RuntimeErrorDemo.java

```
>javac RuntimeErrorDemo.java
>Exit code: 0
>java -cp . RuntimeErrorDemo
Statement to be executed before displaying the fifth element
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: 5
	at RuntimeErrorDemo.main(RuntimeErrorDemo.java:14)
>Exit code: 1
```

Figure 10.4 : Output of the program shown in Figure 10.3

From the output shown in figure 10.4, we can notice that the program execution terminated abruptly from line number 14. The output contains a phrase "ArrayIndexOutOfBoundsException" that indicates the type of exception occurred while executing the program.

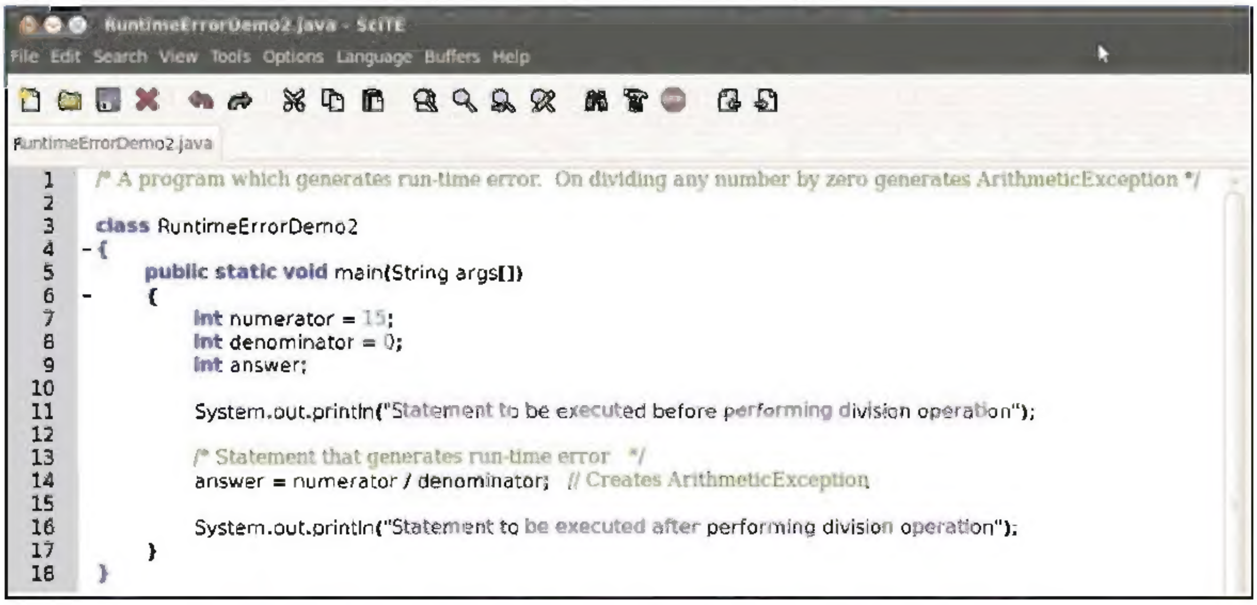
Let us see few other cases that generate exceptions. For each type of exception, there are corresponding Exception classes in Java.

The java.lang and java.io package contains a hierarchy of classes dealing with various exceptions. Few widely observed exceptions are listed in Table 10.1.

Exception Class	**Condition resulting in Exception**	**Example**
ArrayIndexOutOfBoundsException	An attempt to access the array element with an index value that is outside the range of array	int a[] = new int[4]; a[13] = 99;
ArithmeticException	An attempt to divide any number by 0	int a = 50 / 0;
FileNotFoundException	An attempt to access a non-existing file	
NullPointerException	An attempt to use null in a case where an object is required	String s = null; System.out.println(s.length());
NumberFormatException	An attempt to convert string to a number type	String s = "xyz"; int i = Integer.parseInt(s);
PrinterIOException	An I/O error has occurred while printing	

Table 10.1 : List of few widely observed Exceptions

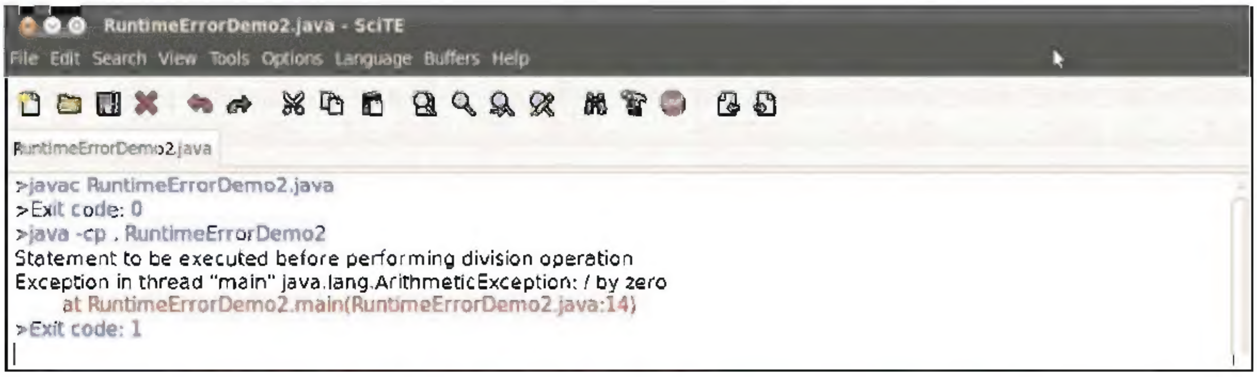
Figure 10.5 shows one more program that generates an exception.


RuntimeErrorDemo2.java - SciTE
File Edit Search View Tools Options Language Buffers Help
RuntimeErrorDemo2.java

```
/* A program which generates run-time error. On dividing any number by zero generates ArithmeticException */

class RuntimeErrorDemo2
{
    public static void main(String args[])
    {
        int numerator = 15;
        int denominator = 0;
        int answer;

        System.out.println("Statement to be executed before performing division operation");

        /* Statement that generates run-time error   */
        answer = numerator / denominator;   // Creates ArithmeticException

        System.out.println("Statement to be executed after performing division operation");
    }
}
```

Figure 10.5 : A program illustrating arithmetic exception

In the program shown in figure 10.5 we try to divide an integer number by zero. A number when divided by zero leads to infinity as a result. Here the program will be compiled successfully but will generate unexpected output due to ArithmeticException. Figure 10.6 shows the output of the execution of program.


RuntimeErrorDemo2.java - SciTE
File Edit Search View Tools Options Language Buffers Help
RuntimeErrorDemo2.java

```
>javac RuntimeErrorDemo2.java
>Exit code: 0
>java -cp . RuntimeErrorDemo2
Statement to be executed before performing division operation
Exception in thread "main" java.lang.ArithmeticException: / by zero
      at RuntimeErrorDemo2.main(RuntimeErrorDemo2.java:14)
>Exit code: 1
```

Figure 10.6 : Output of the code shown in figure 10.5

As we are trying to perform 'division by zero', JVM will terminate the program immediately when it encounters the statement that performs division operation.

Exception Handling

An exception is an error condition. Exception handling is an object-oriented technique for managing errors. While performing exception handling, we try to ensure that the program does not terminate abruptly nor does it generate unexpected output. In this section, we will learn how to handle the exceptions.

Java uses keywords like try, catch and finally to write an exception handler. The keywords try, catch and finally are used in the presence of exceptions, these keywords represent block of statements.

- A try block contains the code that may give rise to one or more exceptions.

- A catch block contains the code that is intended to handle exceptions of a particular type that were created in the associated try block.
- A finally block is always executed before the program ends, regardless of whether any exceptions are generated in the try block or not.

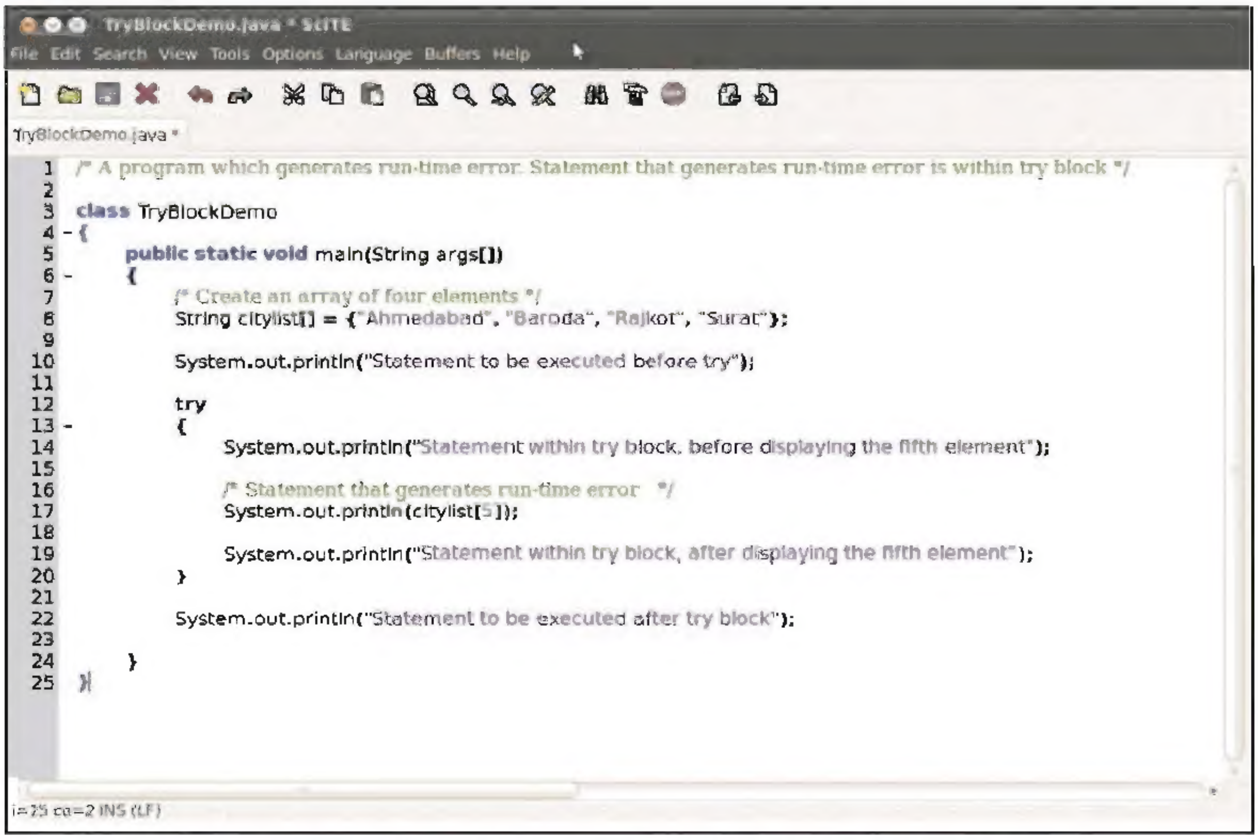
Let us understand these blocks one after another in detail.

The try block

The try statement contains a block of statements within the braces. This is the code that we want to monitor for exceptions. If a problem occurs during its execution, an exception is thrown. Each type of problem (exception) corresponds to an object in Java. A try block may give rise to one or more exceptions. The syntax of the try block is shown below :

```
try
{
   // Set of statements that may generate one or more exceptions
 }
```

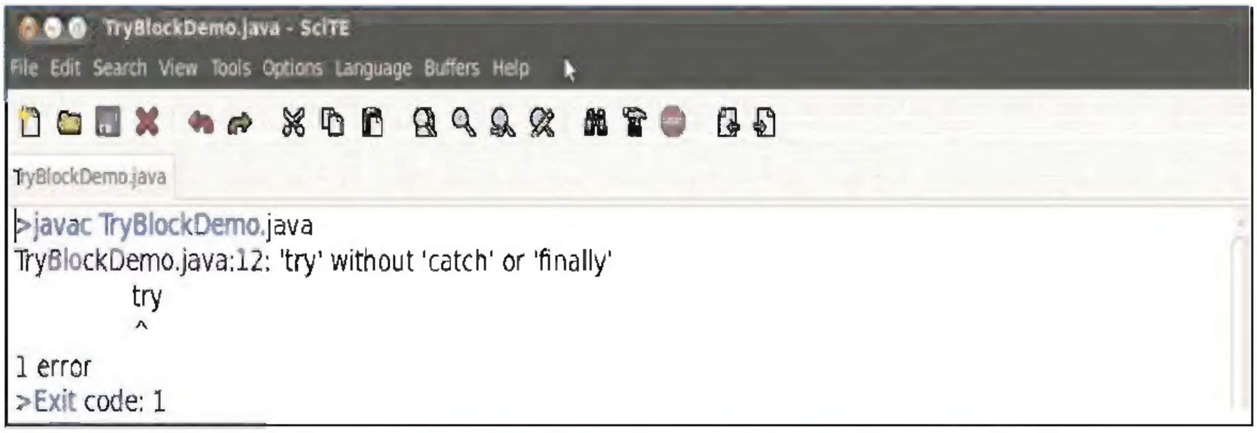
Let us see the code that is placed within a try block; it creates a single exception, which we have already seen earlier. The code shown in figure 10.7, when executed will terminate the program as there is no exception handler. The part of program that may lead to runtime error must be written within the try block.


TryBlockDemo.java * SciTE
File Edit Search View Tools Options Language Buffers Help
TryBlockDemo.java *
i=25 co=2 INS (LF)

```java
/* A program which generates run-time error. Statement that generates run-time error is within try block */

class TryBlockDemo
{
        public static void main(String args[])
        {
                /* Create an array of four elements */
                String citylist[] = {"Ahmedabad", "Baroda", "Rajkot", "Surat"};

                System.out.println("Statement to be executed before try");

                try
                {
                        System.out.println("Statement within try block, before displaying the fifth element");

                        /* Statement that generates run-time error   */
                        System.out.println(citylist[5]);

                        System.out.println("Statement within try block, after displaying the fifth element");
                }

                System.out.println("Statement to be executed after try block");

        }
}
```

Figure 10.7 : A Program that illustrates try block

On compiling the program shown in figure 10.7, it results in compilation error as there has to be either a catch block or a finally block following the try block. The compilation error is shown in figure 10.8.


TryBlockDemo.java - SciTE
File Edit Search View Tools Options Language Buffers Help
TryBlockDemo.java

```
>javac TryBlockDemo.java
TryBlockDemo.java:12: 'try' without 'catch' or 'finally'
          try
          ^
1 error
>Exit code: 1
```

Figure 10.8 : Compilation error in program shown in figure 10.7

The catch block

The catch block must immediately follow the try block. It contains the code that is to be executed to handle an exception. The catch block is an exception handler, for a single try block there can be one or more catch blocks. The syntax of the catch block is shown below:

```
  try
  {
        // Set of statements that may generate one or more exceptions
   }
catch(Exception_Type Exception_object)
  {
         // Code to handle the exception
   }
```

A catch block consists of the keyword catch followed by a single parameter. The code to handle exception has to be written between parentheses. The parameter identifies the type of exception that the block is to deal with. Java supports various types of exceptions, in the later part of this topic we will see how different type of exceptions can be handled by using multiple catch blocks. Figure 10.9 illustrates mechanism of try-catch block.

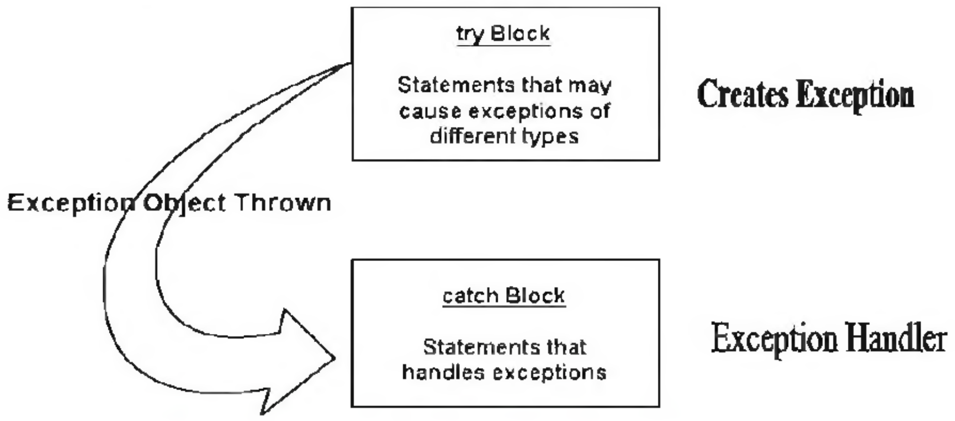
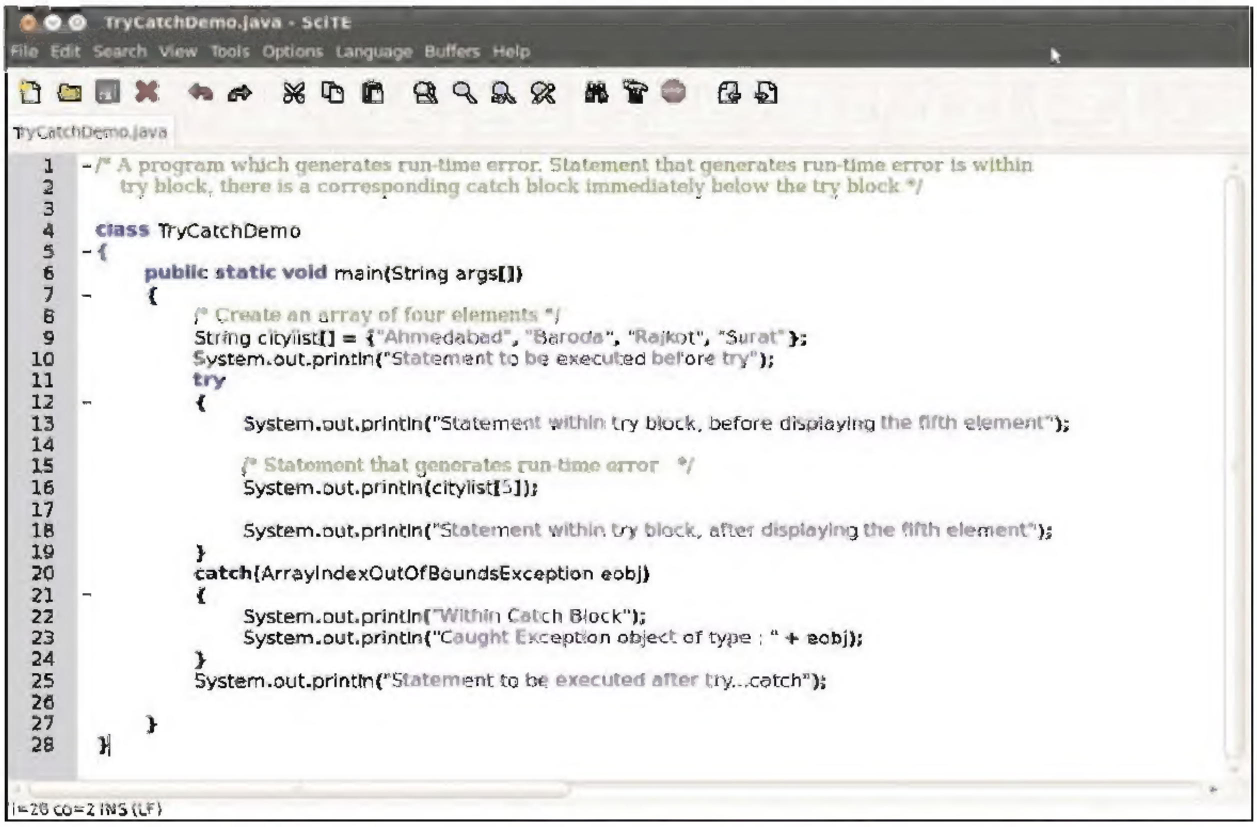

try Block
Statements that may cause exceptions of different types
Creates Exception
Exception Object Thrown
catch Block
Statements that handles exceptions
Exception Handler

Figure 10.9 : Exception Handling Mechanism

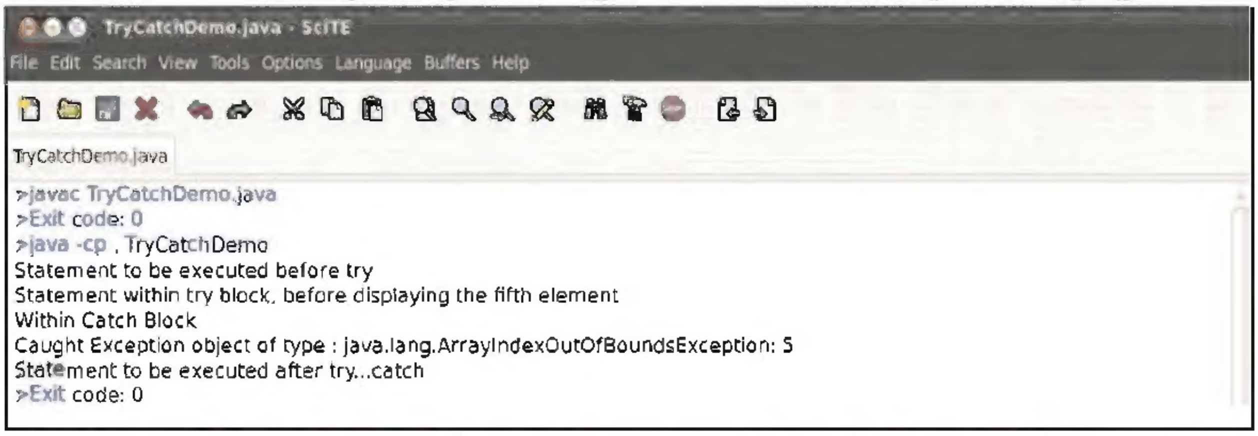
Let us see the example program that uses appropriate exception handler. For the sake of understanding, we will handle ArrayIndexOutOfBounds exception by just catching the object within catch block. In the later sections of the chapter, we will see how the code can be handled to continue the execution of program without termination.


TryCatchDemo.java - SciTE
File Edit Search View Tools Options Language Buffers Help
TryCatchDemo.java
li=28 co=2 INS (LF)

```
1   /* A program which generates run-time error. Statement that generates run-time error is within
2      try block, there is a corresponding catch block immediately below the try block */
3
4   class TryCatchDemo
5   {
6           public static void main(String args[])
7           {
8                   /* Create an array of four elements */
9                   String citylist[] = {"Ahmedabad", "Baroda", "Rajkot", "Surat" };
10                  System.out.println("Statement to be executed before try");
11                  try
12                  {
13                          System.out.println("Statement within try block, before displaying the fifth element");
14
15                          /* Statement that generates run-time error   */
16                          System.out.println(citylist[5]);
17
18                          System.out.println("Statement within try block, after displaying the fifth element");
19                  }
20                  catch(ArrayIndexOutOfBoundsException eobj)
21                  {
22                          System.out.println("Within Catch Block");
23                          System.out.println("Caught Exception object of type : " + eobj);
24                  }
25                  System.out.println("Statement to be executed after try...catch");
26
27          }
28  }
```

Figure 10.10 : A program that illustrates the use of try...catch blocks

The code shown in figure 10.10 will compile successfully and execute. In the program shown in figure 10.10, line 16 contains statement that will generate exception. At line 16, we are trying to access the fifth element of an array citylist[], however the array contains only four members. Our program tries to access array element by specifying index position that is outside the range which leads to an exception. When an exception occurs, an object of type ArrayIndexOutOfBoundsException is created and is thrown; a corresponding catch block handles the exception and does not allow the program to terminate unexpectedly. The catch block contains a reference to object "eobj" which was created and thrown by the try block. Figure 10.11 shows the output of the program.


TryCatchDemo.java - SciTE
File Edit Search View Tools Options Language Buffers Help
TryCatchDemo.java

```
>javac TryCatchDemo.java
>Exit code: 0
>java -cp . TryCatchDemo
Statement to be executed before try
Statement within try block, before displaying the fifth element
Within Catch Block
Caught Exception object of type : java.lang.ArrayIndexOutOfBoundsException: 5
Statement to be executed after try...catch
>Exit code: 0
```

Figure 10.11 : Output of the program shown in figure 10.10

From figure 10.11, we can observe that the program did not terminate in the presence of exception. The statement displaying "after try…catch" gets executed.

Multiple catch blocks

In a single program multiple exceptions can occur. For instance, if we want to upload particular file to a remote computer, it may lead to two distinct exceptions - an exception may occur if the file is not present in our computer or another exception may occur if the computer is not connected to the network. There is a provision in Java to support multiple exceptions. As discussed before, the code that may generate exception should be written within the try block; apart from this there can be multiple catch blocks to handle each type of exception separately.

If the try block throws several different kinds of exceptions, we can write multiple catch blocks, each handling a specific type of exception. This helps the programmer to write separate logic for each type of exception. Figure 10.12 shows the use of multiple catch blocks.

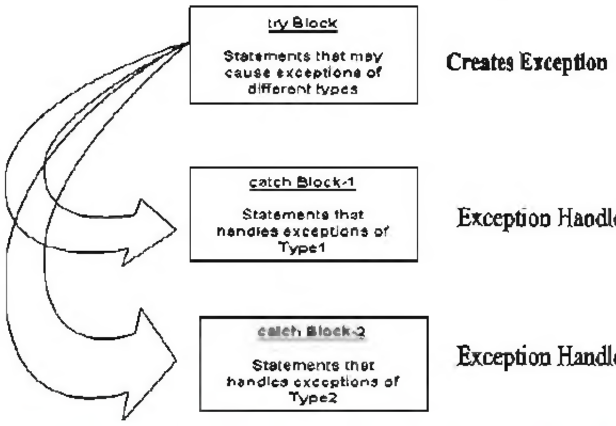
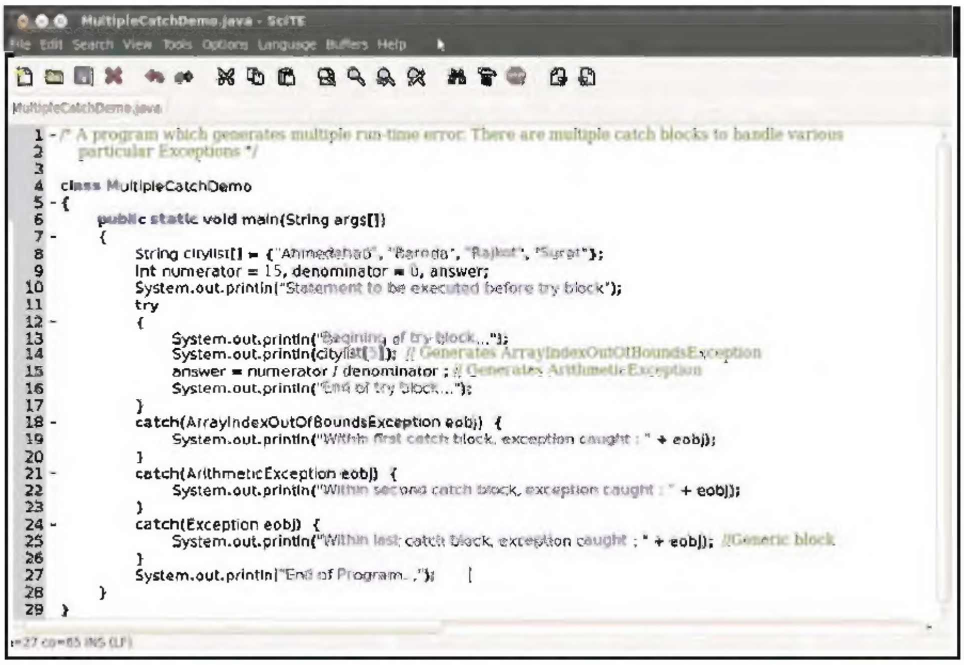

try Block
Statements that may cause exceptions of different types
Creates Exception
catch Block-1
Statements that handles exceptions of Type1
Exception Handler
catch Block-2
Statements that handles exceptions of Type2
Exception Handler

Figure 10.12 : Exception Handling Mechanism with Multiple catch Blocks

Figure 10.13 shows program where multiple catch blocks are used. Here two distinct exceptions are generated within the try block. Exception of type ArrayIndexOutOfBoundsException will be caught by the first catch block and exception of type ArithmeticException will be caught by the second catch block.


MultipleCatchDemo.java - SciTE
File Edit Search View Tools Options Language Buffers Help
MultipleCatchDemo.java

```
/* A program which generates multiple run-time error. There are multiple catch blocks to handle various
    particular Exceptions */

class MultipleCatchDemo
{
        public static void main(String args[])
        {
                String citylist[] = {"Ahmedabad", "Baroda", "Rajkot", "Surat"};
                int numerator = 15, denominator = 0, answer;
                System.out.println("Statement to be executed before try block");
                try
                {
                        System.out.println("Begining of try block...");
                        System.out.println(citylist[5]); // Generates ArrayIndexOutOfBoundsException
                        answer = numerator / denominator ; // Generates ArithmeticException
                        System.out.println("End of try block...");
                }
                catch(ArrayIndexOutOfBoundsException eobj)  {
                        System.out.println("Within first catch block, exception caught : " + eobj);
                }
                catch(ArithmeticException eobj)  {
                        System.out.println("Within second catch block, exception caught : " + eobj);
                }
                catch(Exception eobj)  {
                        System.out.println("Within last catch block, exception caught : " + eobj); //Generic block
                }
                System.out.println("End of Program...");
        }
}
```

li=27 co=65 INS (LF)

Figure 10.13 : A program that illustrates use of multiple catch blocks

The last catch block can handle any type of exception. It is a kind of default catch block and must be the last block when there are multiple catch blocks. While writing program, the order of the specific catch blocks does not matter but the default block has to be placed at the end of all catch blocks. For instance, in the program shown in figure 10.13 we can swap the occurrence of catch blocks starting at line number 18 with the one starting at line number 21; however the catch block given at line 24 must be the last block.

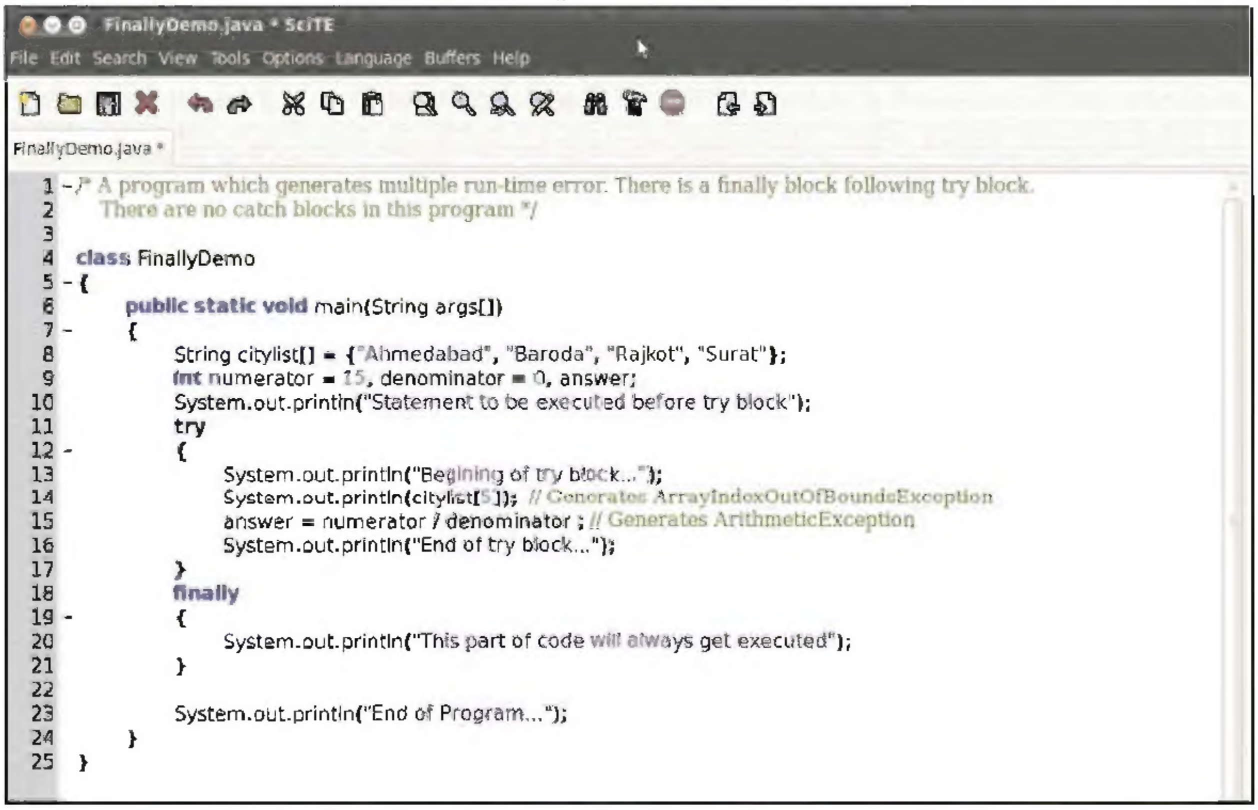
Multiple try blocks can be nested together but care must be taken to write a corresponding catch block for each try block.

The finally block

The finally block is generally used to clean up at the end of executing a try block. We use a finally block when we want to be sure that some particular code is to be run, no matter what exceptions are thrown within the associated try block. A finally block is always executed, regardless of whether or not exceptions are thrown during the execution of the associated try block. A finally block is widely used if a file needs to be closed or a critical resource is to be released at the completion of the program. The syntax of finally block is shown below:

```
finally
 {
    // clean-up code to be executed last
 // statements within this block always get executed even though if run-time errors
        terminate the program abruptly
 }
```

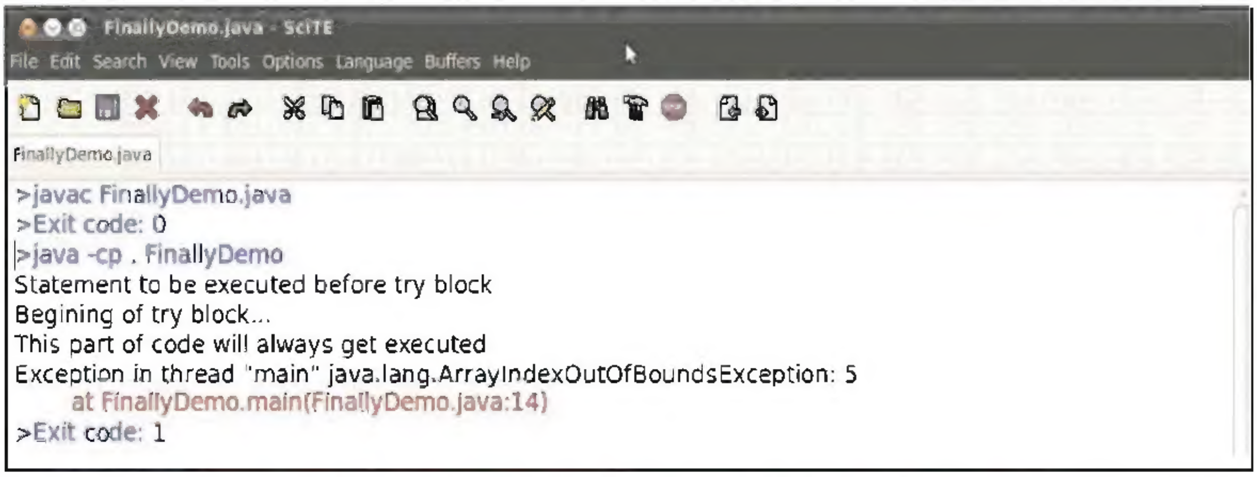
Each try block must always be followed by at least one block that is either a catch block or a finally block. Figure 10.14 shows an example of using finally block.



```
/* A program which generates multiple run-time error. There is a finally block following try block.
   There are no catch blocks in this program */

class FinallyDemo
{
      public static void main(String args[])
      {
            String citylist[] = {"Ahmedabad", "Baroda", "Rajkot", "Surat"};
            int numerator = 15, denominator = 0, answer;
            System.out.println("Statement to be executed before try block");
            try
            {
                  System.out.println("Begining of try block...");
                  System.out.println(citylist[5]);  // Generates ArrayIndexOutOfBoundsException
                  answer = numerator / denominator ; // Generates ArithmeticException
                  System.out.println("End of try block...");
            }
            finally
            {
                  System.out.println("This part of code will always get executed");
            }

            System.out.println("End of Program...");
      }
}
```

Figure 10.14 : Program which illustrates finally block without catch block

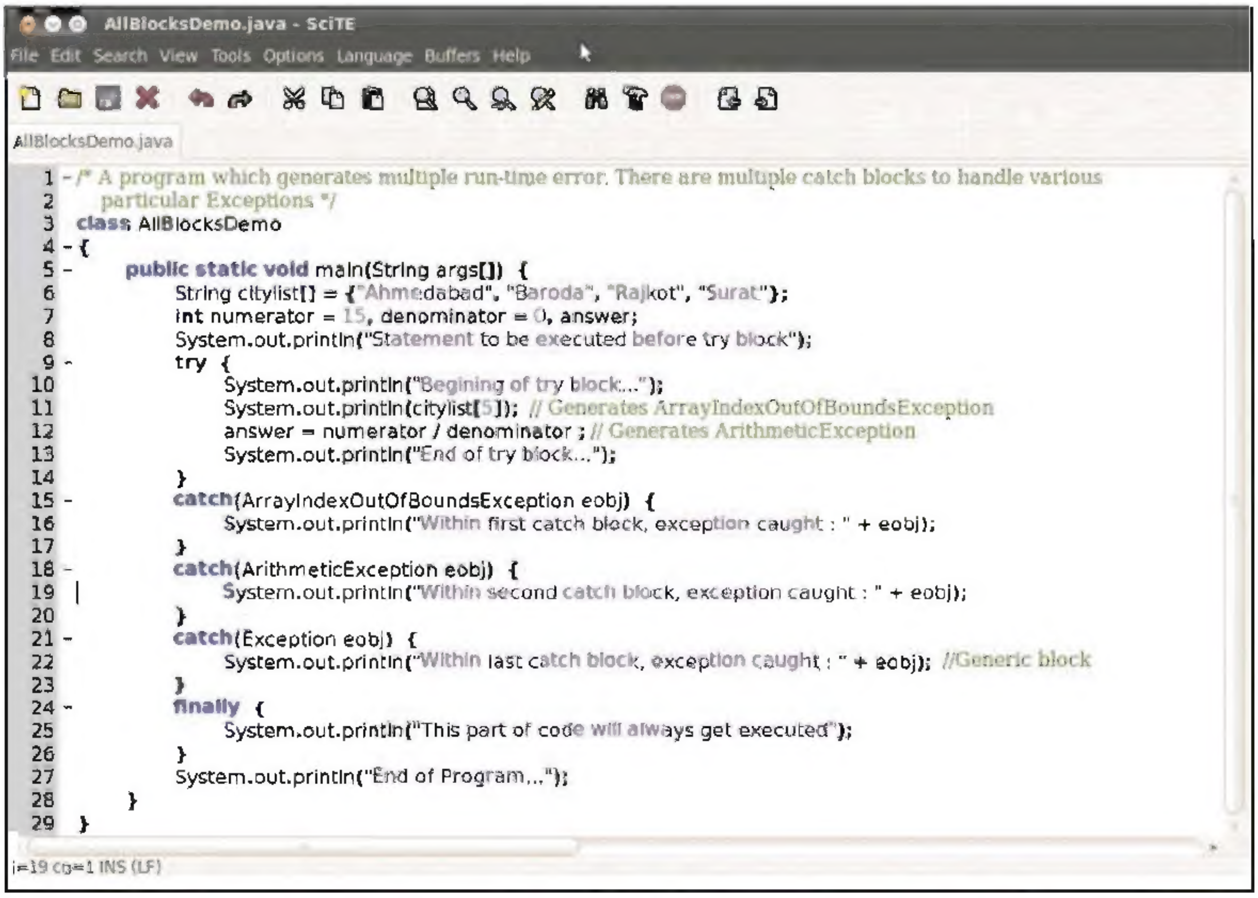
In the program shown in figure 10.14, as there is no catch block, the program terminates abruptly due to the exception generated at line 14; statements present in line 15 and line 16 will not be executed. However, in the presence of finally block, the program executes the statements within the finally block before being terminated. The output of program is shown in figure 10.15.



```
>javac FinallyDemo.java
>Exit code: 0
>java -cp . FinallyDemo
Statement to be executed before try block
Begining of try block...
This part of code will always get executed
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: 5
        at FinallyDemo.main(FinallyDemo.java:14)
>Exit code: 1
```

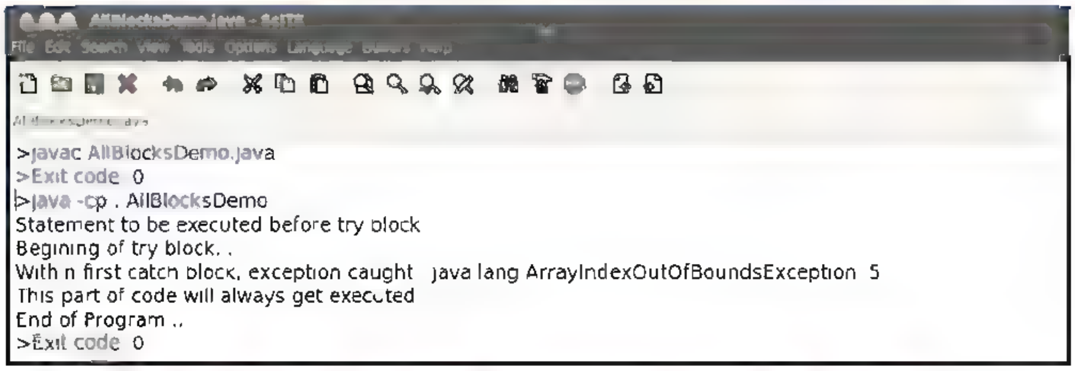
Figure 10.15 : The output of program shown in figure 10.14

Let us look at an example with multiple catch blocks and a finally block all used together. The program shown in figure 10.16 contains all these blocks. It is a complete program with multiple catch blocks for corresponding exceptions being generated within the try block. Output of program is shown in figure 10.17.


AllBlocksDemo.java - SciTE
File Edit Search View Tools Options Language Buffers Help
AllBlocksDemo.java
li=19 co=1 INS (LF)

```java
/* A program which generates multiple run-time error. There are multiple catch blocks to handle various
   particular Exceptions */
class AllBlocksDemo
{
    public static void main(String args[]) {
        String citylist[] = {"Ahmedabad", "Baroda", "Rajkot", "Surat"};
        int numerator = 15, denominator = 0, answer;
        System.out.println("Statement to be executed before try block");
        try {
            System.out.println("Begining of try block...");
            System.out.println(citylist[5]); // Generates ArrayIndexOutOfBoundsException
            answer = numerator / denominator ; // Generates ArithmeticException
            System.out.println("End of try block...");
        }
        catch(ArrayIndexOutOfBoundsException eobj) {
            System.out.println("Within first catch block, exception caught : " + eobj);
        }
        catch(ArithmeticException eobj) {
            System.out.println("Within second catch block, exception caught : " + eobj);
        }
        catch(Exception eobj) {
            System.out.println("Within last catch block, exception caught : " + eobj); //Generic block
        }
        finally {
            System.out.println("This part of code will always get executed");
        }
        System.out.println("End of Program...");
    }
}
```

Figure 10.16 : Program illustrating try, catch and finally blocks



```
>javac AllBlocksDemo.java
>Exit code  0
>java -cp . AllBlocksDemo
Statement to be executed before try block
Begining of try block. .
With n first catch block, exception caught  java lang ArrayIndexOutOfBoundsException  5
This part of code will always get executed
End of Program ..
>Exit code  0
```

Figure 10.17 : Output of the program shown in figure 10.16

From the output it is clear that the control of program execution switched from line 11 to the first catch block, later it switched to the finally block. The last two catch blocks did not execute. Although the program did not execute completely, it terminated gracefully.

A finally block is associated with a particular try block, and it must be located immediately following any catch blocks for the corresponding try block. If there are no catch blocks, then the finally block can be positioned immediately after the try block. If the finally block or catch blocks are not positioned correctly, then the program will not compile.

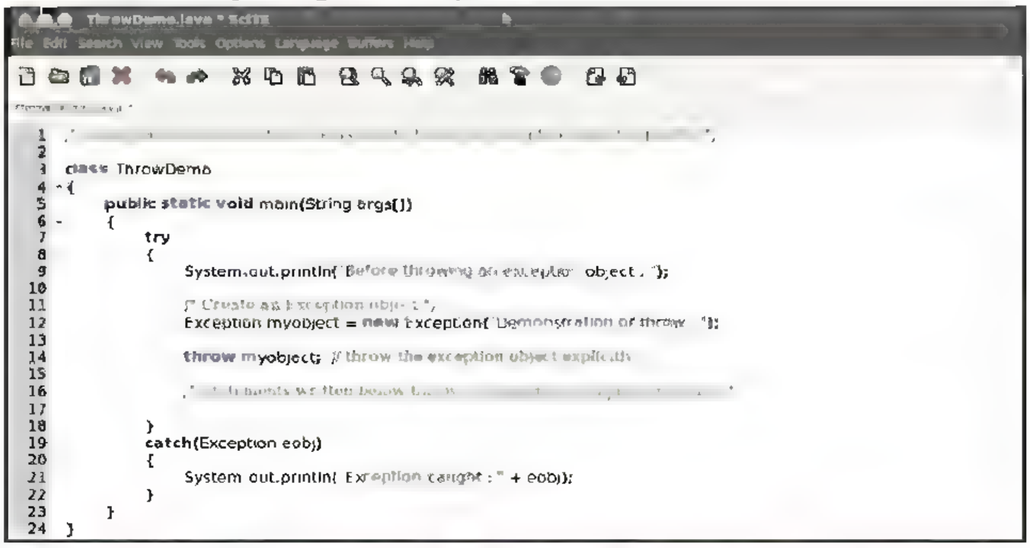
The throw statement

The throw keyword is used to explicitly throw an Exception object. In the example programs that we have seen so far, the JVM created an exception object and was throwing it automatically. For example, an object of ArithmeticException was created when we tried to perform a divide by zero operation and it was thrown automatically by the JVM.

Java does provide mechanism to create an Exception object and throw it explicitly. The object that we throw must be of type java.lang.Throwable, (object of Throwable class or any of its sub-classes) otherwise a compile error occurs. The syntax to throw an exception object is as follows:

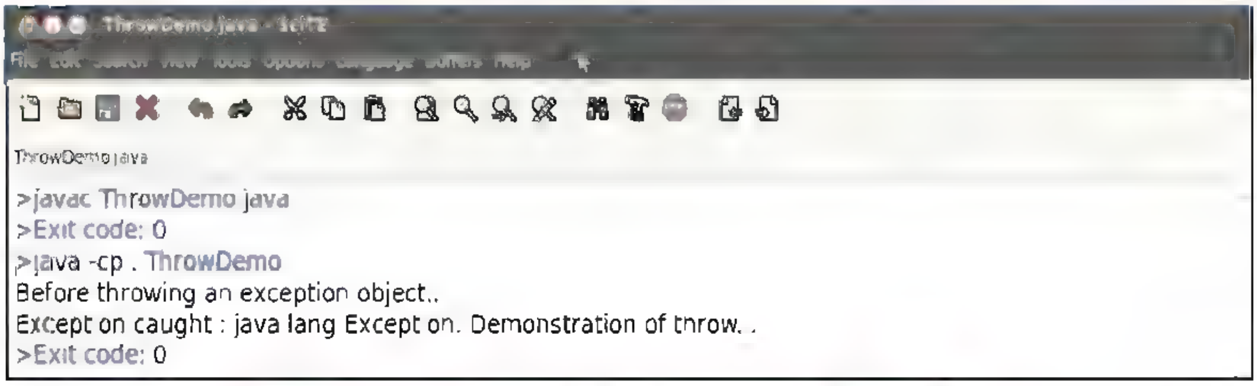
throw exception_object;

When a throw statement is encountered, a search for matching catch block begins. Any subsequent statements in the try or catch block are not executed. The code in figure 10.18 shows the use of throw statement, its output is given in figure 10.19.



```
/* [illegible] */

class ThrowDemo
{
	public static void main(String args[])
	{
		try
		{
			System.out.println("Before throwing an exception object . ");

			/* Create an Exception object */
			Exception myobject = new Exception("Demonstration of throw  ");

			throw myobject; // throw the exception object explicitly

			/* statements written below [illegible] */

		}
		catch(Exception eobj)
		{
			System.out.println("Exception caught : " + eobj);
		}
	}
}
```

Figure 10.18 : Program which illustrates the use of throw keyword



```
ThrowDemo.java
>javac ThrowDemo.java
>Exit code: 0
>java -cp . ThrowDemo
Before throwing an exception object..
Exception caught : java.lang.Exception: Demonstration of throw..
>Exit code: 0
```

Figure 10.19 : Output of the program in figure 10.18

In the program shown in figure 10.18, we have created an object "obj" of the class Exception, the same object is thrown using throw statement. There has to be catch blocks to handle the exception object thrown explicitly.

The throws Clause

We have explored the try-catch-finally blocks, the programs we discussed so far were simple programs that didn't involve the use of methods. Few questions may arise like what will happen if an exception occurs in a method or a constructor, where will we place the try-catch blocks. There are two alternate approaches to handle exceptions created by a method :

- Write a try-catch block within the method or a constructor that may generate an exception
- Invoking a method (that may generate exception) or constructor within a try block

A throws clause can be used in a method declaration or constructor declaration to inform that the code within the constructor or method may throw an Exception. It also signifies that there is no catch block within the method that can handle the exception. When we write a constructor or a method that can throw exceptions to its caller, it is useful to document that fact. The throws keyword is used with the declaration of method.

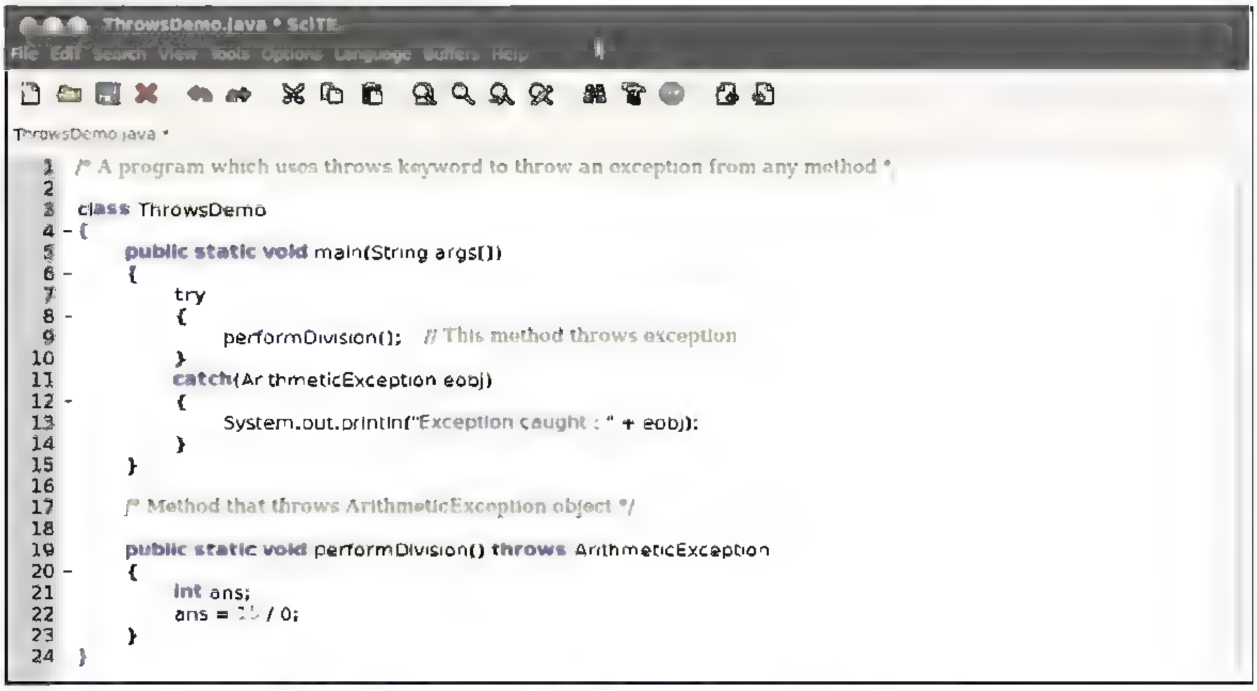
A throws clause can be used in a method declaration as follows :

```
method_Modifiers return_type method_Name(parameters) throws Exception list... {
.....
// body of the method
.....
}
```

A method can throw multiple exceptions. Each type of exception that a method can throw must be stated in the method header. For example, a method header can be like:

```
performDivision() throws ArithmeticException, ArrayIndexOutOfBoundsException
{
.....
// body of the method
.....
}
```

The program in figure 10.20 demonstrates the use of throws clause in the presence of user defined methods. In the following code, it must be noted that if the method throws an Exception object, there must be a matching catch handler. However if we are catching an exception type within the method, there is no need to throw it.



```
/* A program which uses throws keyword to throw an exception from any method *

class ThrowsDemo
{
    public static void main(String args[])
    {
        try
        {
            performDivision();   // This method throws exception
        }
        catch(ArithmeticException eobj)
        {
            System.out.println("Exception caught : " + eobj);
        }
    }

    /* Method that throws ArithmeticException object */

    public static void performDivision() throws ArithmeticException
    {
        int ans;
        ans = 25 / 0;
    }
}
```

Figure 10.20 : Program which illustrates the use of throws keyword

In the program of figure 10.20, we have written a method performDivision(), an ArithmeticException object will be generated in this method. The method performDivision() is called in the main() method of our program. It is quite obvious that as there is no exception-handling mechanism within the performDivision() method, the exception will be caught by calling method and there has to be a handler in the calling method. Similarly, there can be exceptions within the Constructors of Java class.

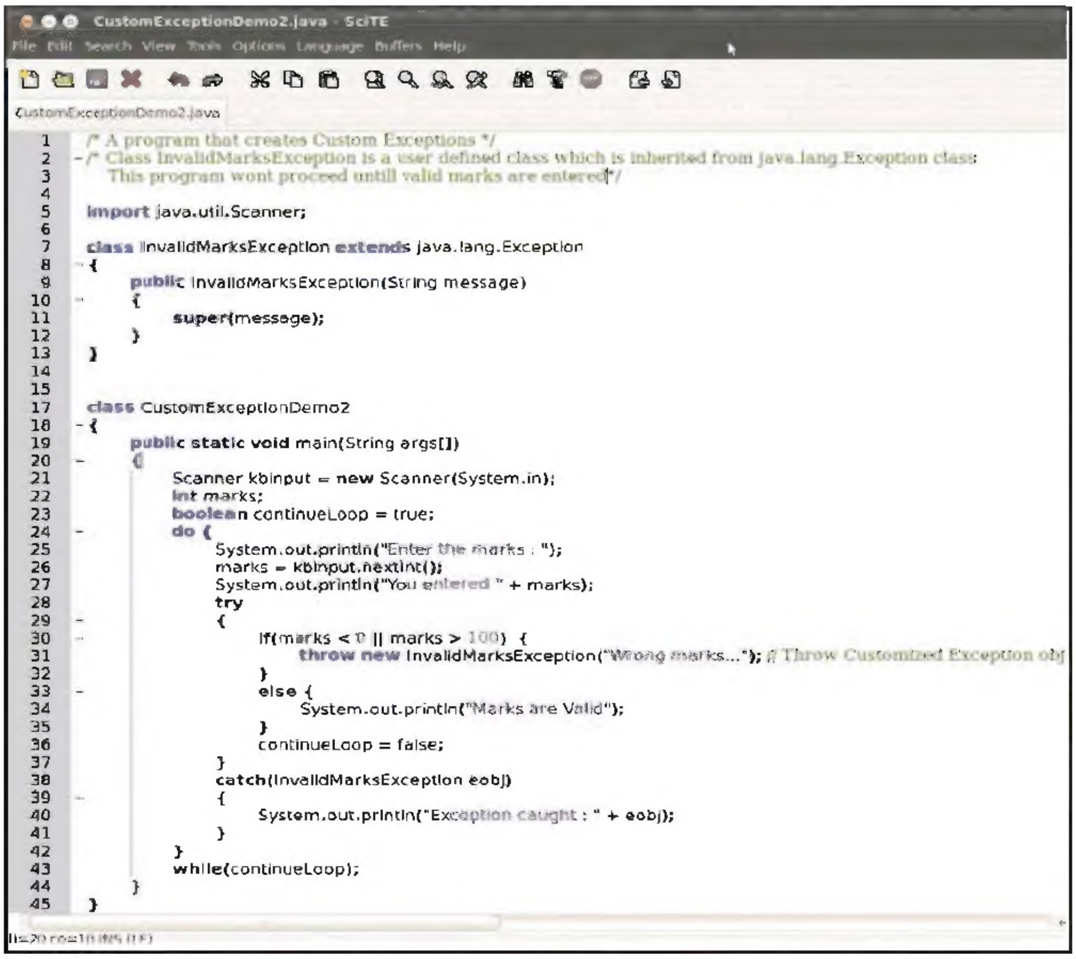
Creating Custom Exceptions

Java allows creating our own exception classes according to application-specific problems. For instance, we are writing a program to generate a mark sheet in which the user is asked to enter marks of different subjects. The marks must be in the range of 0 to 100, suppose if the user enters negative marks or the value is above 100 then the program must generate an Exception. Such kind of exceptions are application specific, Java does not provide built-in exception classes for application specific exceptions.

We can create user-defined exceptions by creating a subclass of Exception class. These exceptions can be thrown explicitly using the throws statement. However it is required to catch this exception and handle it accordingly. Let us see a program code that creates a custom exception to validate the marks.

The program shown in figure 10.21 accepts input from the user. We have used java.util.Scanner class to accept input from the keyboard in line number 21. The "nextInt()" method of the Scanner class helps in reading integer input from the console. We will discuss the functionality of Scanner class in the next chapter that deals with Files and I/O.

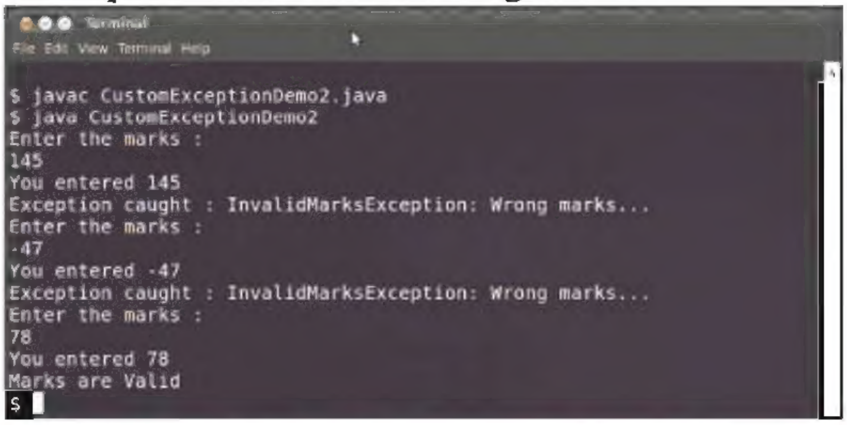
Here we have implemented two classes. An additional class is required to create a custom exception. The class "InvalidMarksException" extends Exception class of java.lang package; it contains a single parameter constructor that accepts string to describe the type of error.



```java
/* A program that creates Custom Exceptions */
/* Class InvalidMarksException is a user defined class which is inherited from java.lang.Exception class;
   This program wont proceed untill valid marks are entered*/

import java.util.Scanner;

class InvalidMarksException extends java.lang.Exception
{
	public InvalidMarksException(String message)
	{
		super(message);
	}
}


class CustomExceptionDemo2
{
	public static void main(String args[])
	{
		Scanner kbinput = new Scanner(System.in);
		int marks;
		boolean continueLoop = true;
		do {
			System.out.println("Enter the marks : ");
			marks = kbinput.nextInt();
			System.out.println("You entered " + marks);
			try
			{
				if(marks < 0 || marks > 100)  {
					throw new InvalidMarksException("Wrong marks..."); // Throw Customized Exception obj
				}
				else {
					System.out.println("Marks are Valid");
				}
				continueLoop = false;
			}
			catch(InvalidMarksException eobj)
			{
				System.out.println("Exception caught : " + eobj);
			}
		}
		while(continueLoop);
	}
}
```

Figure 10.21 : User defined exception class

In the main method, the business logic is coded (application specific) that ensures whether the marks are in range or not. If the marks are not in range, we create an object of type "InvalidMarksException" and throw it. There has to be a catch block to handle this exception. This program will not proceed unless valid marks are entered. In case if the user enters invalid marks, he/she is asked to keep on re-entering until the input is correct. It must be noted that try-catch blocks are used within a do-while loop. The output of the code is shown in figure 10.22.



```
$ javac CustomExceptionDemo2.java
$ java CustomExceptionDemo2
Enter the marks :
145
You entered 145
Exception caught : InvalidMarksException: Wrong marks...
Enter the marks :
-47
You entered -47
Exception caught : InvalidMarksException: Wrong marks...
Enter the marks :
78
You entered 78
Marks are Valid
$
```

Figure 10.22 : Output of the program shown in figure 10.21

Note: SciTE is an editor to type the programs. If the program accepts data from the keyboard, it is advisable to execute the program at command prompt; however, a simple program that does not require user interaction can be executed from within SciTE editor.

Advantages of Exception Handling

Throughout the chapter, we have advocated the use of exception handling in Java programs. By now, it must be clear that a good program must always handle exceptions rather than the program being terminated abruptly. Let us briefly look at the advantages of using exception handling in our Java programs. Few advantages of using exception-handling in Java programs are listed below:

- It allows us to maintain normal flow of program. In the absence of exception handling, the flow of program is disturbed.
- It allows writing separate error handling code from the normal code.
- Error types can be grouped and differentiated within the program.
- Assertions can be used to debug the program before deploying it to the clients.
- It provides an easy mechanism to log various run-time errors while executing the program.

Summary

In chapter we learnt that a program can use exceptions to indicate that an error occurred. A program can catch exceptions by using a combination of the try, catch, and finally blocks.

To throw an exception, we use the throw statement and provide it with an exception object (subclass of java.lang.Throwable class). A method that throws an uncaught, checked exception must include a throws clause in its declaration. The try statement should contain at least one catch block or a finally block and may have multiple catch blocks. Assertions are used to check that something should never happen; they are used by programmers for debugging purpose.

EXERCISE

1. What is the difference between compile-time error and run-time error ?
2. What is an Exception ? Give examples of few Exceptions found in Java.
3. How are the exceptions handled in java ?
4. What is the significance of try, catch and finally block ?
5. What is the use of throw and throws keyword ?
6. How do you create a custom Exception ?
7. Choose the most appropriate option from those given below :

   (1) Which of the following refers to an error condition in object-oriented programming terminology ?

(a) anomaly	(b) abbreviation
(c) exception	(d) deviation

(2) Which of the following is a correct word for all Java Exceptions ?

(a) Errors (b) Runtime Exceptions

(c) Throwables (d) Omissions

(3) Which of the following statements is true ?

(a) Exceptions are more serious than Errors.

(b) Errors are more serious than Exceptions.

(c) Errors and Exceptions are equally serious.

(d) Exceptions and Errors are the same thing.

(4) Which of the following elements is not included in try block ?

(a) the keyword try (b) the keyword catch

(c) the curly braces (d) statements that might cause Exceptions

(5) Which of the following block handles or takes appropriate action when an Exception occurs ?

(a) try (b) catch

(c) throws (d) handles

(6) Which of the following should be within a catch block ?

(a) finally block (b) single statement that handles Exception

(c) any number of statements to handle Exception

(d) throws keyword

(7) What will happen when a try block does not generate an Exception and you have included multiple catch blocks ?

(a) they all execute (b) only the first matching one executes

(c) no catch block executes (d) only the first catch block executes

(8) Which of the following is an advantage of using a try…catch block ?

(a) Exceptional events are eliminated

(b) Exceptional events are reduced

(c) Exceptional events are integrated with regular events

(d) Exceptional events are isolated from regular events

(9) Which of the following methods can throw an Exception ?

(a) methods with throws clause (b) methods with a catch block

(c) methods with a try block (d) methods with finally block

(10) Which of the following is least important to know if you want to be able to use a method to its full potential ?

(a) the method's return type

(b) the type of arguments the method requires

(c) the number of statements within the method

(d) the type of Exceptions the method throws

LABORATORY EXERCISE

1. Write a java program that uses a method - "add()", to add the elements in an array, include a try block within the method, so that the method must handle ArrayIndexOutOfBoundsException.

2. In the above example, remove the try...catch block from the method. By using the throws keyword, the method that invokes "add()" method must handle the exception.

3. Write a java program that throws and catches an ArithmeticException when you attempt to take the square root of a negative value. Prompt the user for an input value and try the Math.sqrt() method on it. The application either displays the square root or catches the thrown Exception and displays an appropriate message. Save the file as SqrtException.java.

4. Write a java program to validate the birth date. Create a custom exception "InvalidBirthDateException". Ask the user to enter any date, if the birth date is after the current date then throw the "InvalidBirthDateException", also write a suitable code to handle such Exceptions.

5. Write a java program to simulate bank transactions. Take two variables, balanceAmount and withdrawAmount. Program must assert if the withdrawAmount is less than balanceAmount.

6. Write a java program to simulate bank transactions. Take two variables, balanceAmount and withdrawAmount. Create a custom exception, "InvalidTransaction", your program must throw the "InvalideTransaction" exception if the withdrawAmount is less than balanceAmount. Write appropriate handlers for this Exception.

7. Write a java program to validate the birth date. Create three different custom exceptions like "InvalidDateException", "InvalidMonthException" and "InvalidYearException". Throw "InvalidDateException" if the date is negative or greater than 30, throw "InvalidMonthException" if the month is negative or greater than 12, throw "InvalidYearException" if the year is below 1950 or after the current year.